Algorithms in Computational Biology

More on BWT

Today: Last class!

- Please don't forget to submit final implementation project/HW by next week
  (via email, or share git repo)

  Note: I would like a readme/overview!

  Discuss design designs, how you tested (~~&~~ show me some tests), how to compile/use, and any comparison's or lessons learned.

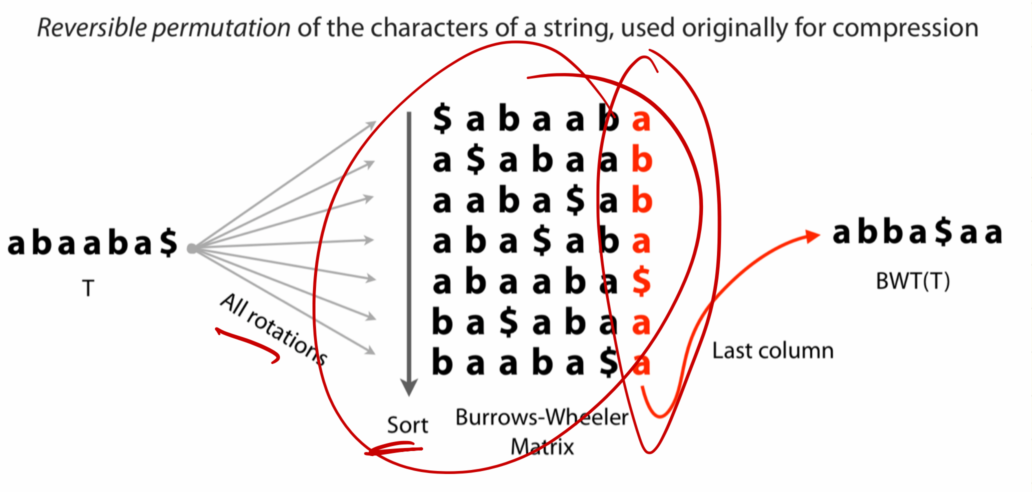
- All HW is graded! ☺

  Come get HWS next week.

  – Don't forget instructor evaluations

Today: More on BWT

First, recap:



Reversible permutation of the characters of a string, used originally for compression

```
$ a b a a b a
a $ a b a a b
a a b a $ a b
a b a $ a b a
a b a a b a $
b a $ a b a a
b a a b a $ a
```

Key points:

- Compressible
- Reversible
- Useful (& fast) for searching

Reversing: → abba $aa

Sort:

Sort	BWT
$	a
a	b
a	b
a	a
a	$
b	a
b	a

⇒

all pairs:

a$
ba
ba
aa
→ $a
ab
ab

sort again:

sort	BWT
$a	a
a$	b
aa	b
ab	a
ab	$
ba	a
ba	a

⇒

all triples:

a$a
ba$
baa
aab
$ab
aba
aba

sort:

Sort	BWT
	a
	b
	b
	a
	$
	a
	a

⇒

all 4-tuples:

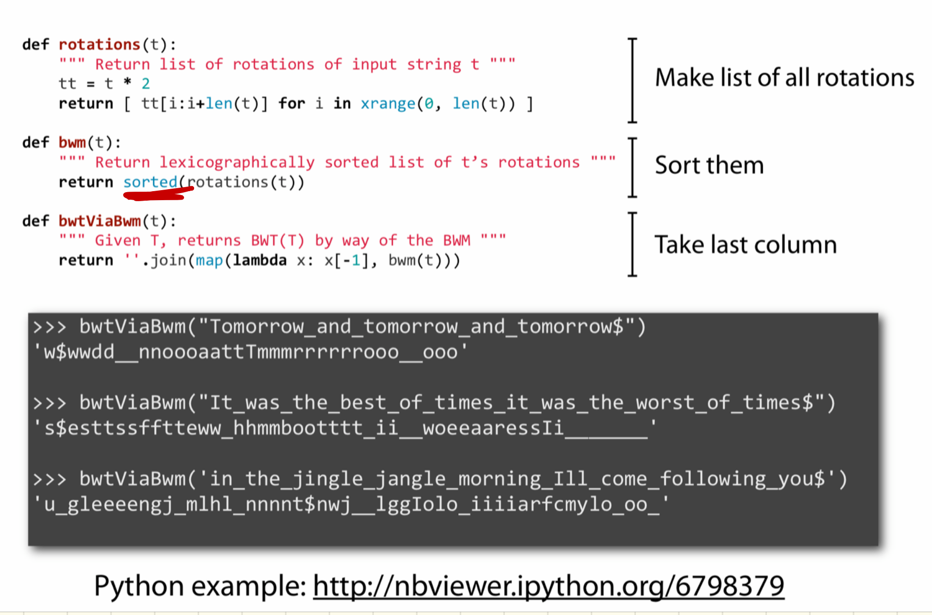
sort:	sorted	BWT		5-tuples
		a		
		b		
		b	⇒	
		a		
		$		
		a		
		a		

sort:	sorted	BWT		6 tuples
		a		
		b	⇒	
		b		
		a		
		$		
		a		
		a		

sort:	sorted	BWT		7-tuples
		a		
		b	⇒	
		b		
		a		
		$		
		a		
		a		

original: row ending in $

Code: easy (if slow)



```python
def rotations(t):
    """ Return list of rotations of input string t """
    tt = t * 2
    return [ tt[i:i+len(t)] for i in xrange(0, len(t)) ]

def bwm(t):
    """ Return lexicographically sorted list of t's rotations """
    return sorted(rotations(t))

def bwtViaBwm(t):
    """ Given T, returns BWT(T) by way of the BWM """
    return ''.join(map(lambda x: x[-1], bwm(t)))
```

Make list of all rotations

Sort them

Take last column

```
>>> bwtViaBwm("Tomorrow_and_tomorrow_and_tomorrow$")
'w$wwdd__nnoooaattTmmmrrrrrrooo__ooo'

>>> bwtViaBwm("It_was_the_best_of_times_it_was_the_worst_of_times$")
's$esttssfftteww_hhmmbootttt_ii__woeeaaressIi_______'

>>> bwtViaBwm('in_the_jingle_jangle_morning_Ill_come_following_you$')
'u_gleeeengj_mlhl_nnnnt$nwj__lggIolo_iiiiarfcmylo_oo_'
```

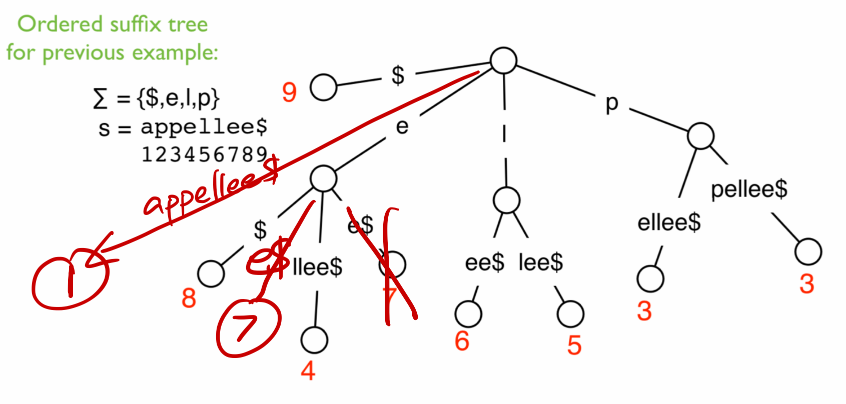
Python example: http://nbviewer.ipython.org/6798379

Runtime? $O(n^2 \log n)$

(Important takeaway — ☆
1 line of code is
not $O(1)$ time!)

Last time:

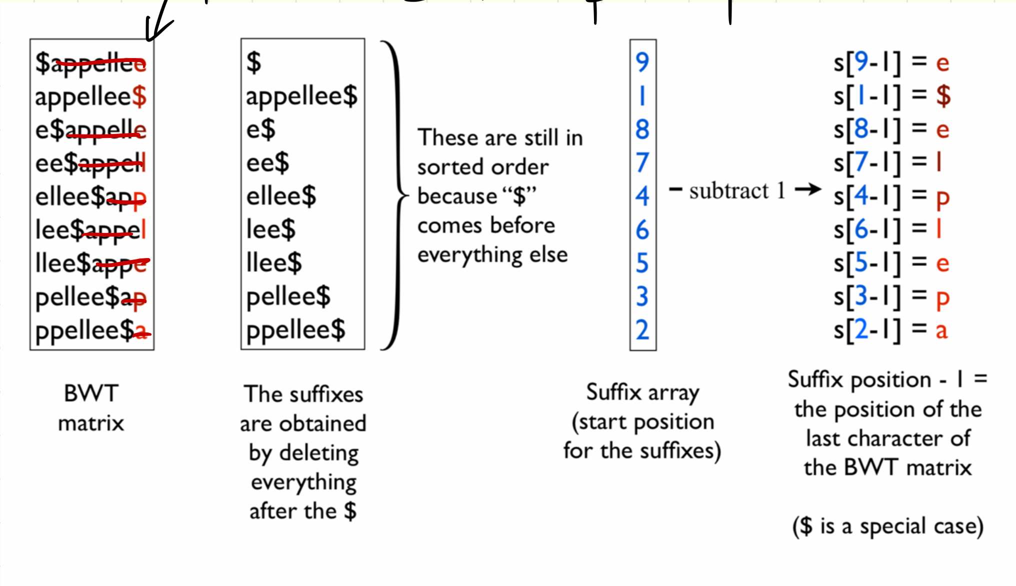
Connection to suffix trees + suffix arrays:



Ordered suffix tree for previous example:

Σ = {$,e,l,p}
s = appellee$
123456789

BWT: e$elplepa



$~~appellee~~	$	9	s[9-1] = e
appellee$	appellee$	1	s[1-1] = $
e$~~appelle~~	e$	8	s[8-1] = e
ee$~~appell~~	ee$	7	s[7-1] = l
ellee$~~app~~	ellee$	4	s[4-1] = p
lee$~~appel~~	lee$	6	s[6-1] = l
llee$~~appe~~	llee$	5	s[5-1] = e
pellee$~~ap~~	pellee$	3	s[3-1] = p
ppellee$~~a~~	ppellee$	2	s[2-1] = a

These are still in sorted order because "$" comes before everything else

– subtract 1 →

BWT matrix

The suffixes are obtained by deleting everything after the $

Suffix array (start position for the suffixes)

Suffix position - 1 = the position of the last character of the BWT matrix

($ is a special case)

How to reverse more efficiently?

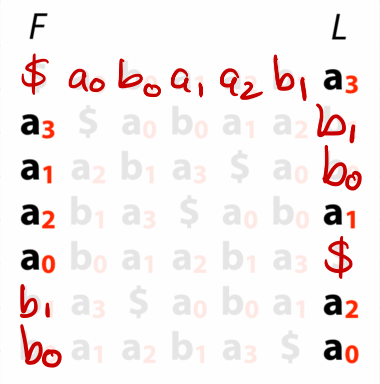
Today: LF Mappings

Give each character a T-rank := # of times character appeared previously in string

Ex: $a_0\, b_0\, a_1\, a_2\, b_1\, a_3\, \$$

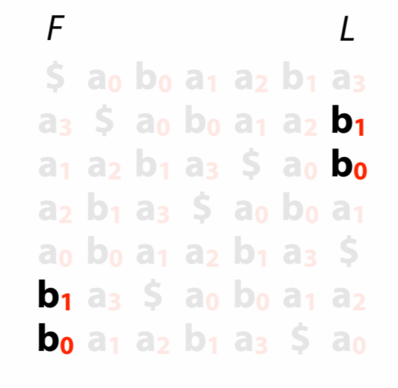
Why? Look back at BWT:

Key fact: relative order is same in F + L (of T-ranks)



F						L
$	a_0	b_0	a_1	a_2	b_1	a_3
a_3	$	a_0	b_0	a_1	a_2	b_1
a_1	a_2	b_1	a_3	$	a_0	b_0
a_2	b_1	a_3	$	a_0	b_0	a_1
a_0	b_0	a_1	a_2	b_1	a_3	$
b_1	a_3	$	a_0	b_0	a_1	a_2
b_0	a_1	a_2	b_1	a_3	$	a_0

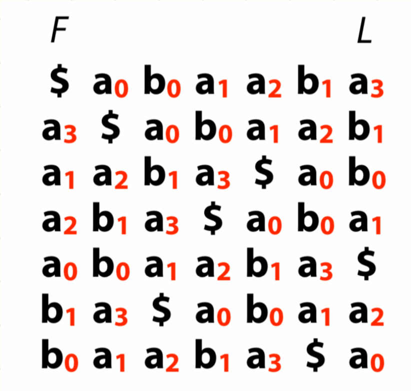
This is true for any value:



F						L
\$	a_0	b_0	a_1	a_2	b_1	a_3
a_3	\$	a_0	b_0	a_1	a_2	**b_1**
a_1	a_2	b_1	a_3	\$	a_0	**b_0**
a_2	b_1	a_3	\$	a_0	b_0	a_1
a_0	b_0	a_1	a_2	b_1	a_3	\$
b_1	a_3	\$	a_0	b_0	a_1	a_2
b_0	a_1	a_2	b_1	a_3	\$	a_0

Called LF-mapping:

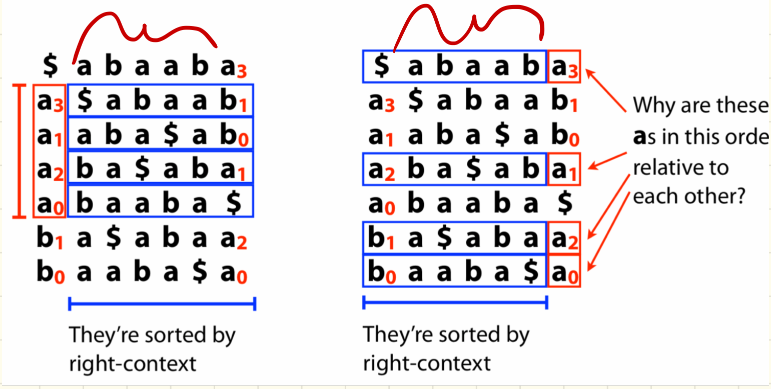
The i^{th} occurance of character c in L and character c in R always correspond to same occurance in original string.



F						L
\$	a_0	b_0	a_1	a_2	b_1	a_3
a_3	\$	a_0	b_0	a_1	a_2	b_1
a_1	a_2	b_1	a_3	\$	a_0	b_0
a_2	b_1	a_3	\$	a_0	b_0	a_1
a_0	b_0	a_1	a_2	b_1	a_3	\$
b_1	a_3	\$	a_0	b_0	a_1	a_2
b_0	a_1	a_2	b_1	a_3	\$	a_0

Why??

Because we're doing lexicographical (ie alphabetical) sorted order!



$\$\ a\ b\ a\ a\ b\ a_3$

$a_3\ \$\ a\ b\ a\ a\ b_1$

$a_1\ a\ b\ a\ \$\ a\ b_0$

$a_2\ b\ a\ \$\ a\ b\ a_1$

$a_0\ b\ a\ a\ b\ a\ \$$

$b_1\ a\ \$\ a\ b\ a\ a_2$

$b_0\ a\ a\ b\ a\ \$\ a_0$

They're sorted by right-context

$\$\ a\ b\ a\ a\ b\ a_3$

$a_3\ \$\ a\ b\ a\ a\ b_1$

$a_1\ a\ b\ a\ \$\ a\ b_0$

$a_2\ b\ a\ \$\ a\ b\ a_1$

$a_0\ b\ a\ a\ b\ a\ \$$

$b_1\ a\ \$\ a\ b\ a\ a_2$

$b_0\ a\ a\ b\ a\ \$\ a_0$

They're sorted by right-context

Why are these **a**s in this orde relative to each other?

All the a's have same order.

Ties broken by same sorted string — it's suffix of one + prefix of other!

Sometimes called "First-Last property".

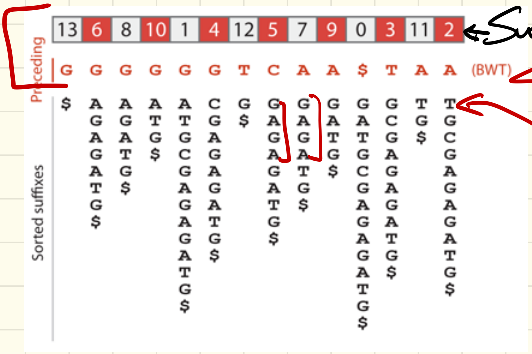
Now: How can we use BWT to look for all repeats of one string?

Let's look at a biological data set:

String: GATGCCGAGAGATG$

5 7

Compute all cyclic permutations (or do suffix array from last time)



	13	6	8	10	1	4	12	5	7	9	0	3	11	2	
Preceding	G	G	G	G	G	G	T	C	A	A	$	T	A	A	(BWT)
Sorted suffixes	$	AGAGATG$	AGATG$	ATG$	ATGCGAGAGATG$	CGAGAGATG$	G$	GAGAGATG$	GAGATG$	GATG$	GATGCGAGAGATG$	GCGAGAGATG$	TG$	TGCGAGAGATG$	

Suffix array

BWT

F

Let's look for all "GAGA" in text.

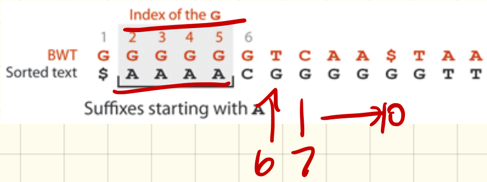
Counting & backward search:

All GAGA end with "A".

Each of these "A"s is 1st letter of some suffix.

However, only suffixes preceded by a G can be options.

BWT stores this!



Index of the G

	1	2	3	4	5	6								
BWT	G	G	G	G	G	G	T	C	A	A	$	T	A	A
Sorted text	$	A	A	A	A	C	G	G	G	G	G	G	T	T

Suffixes starting with A

These must be stored next to each other in suffix array (since all start the same).

Q: Where is the 1st G in the string?

(Remember – descending order)

Since 1st G in 6, these are 7-10

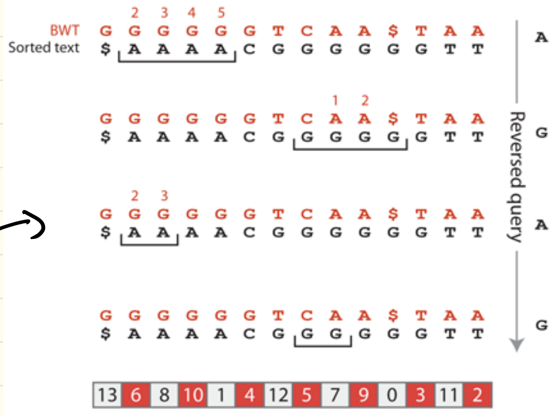
So: we continue.

Looking for GAGA, so "A" must come before "GA".

In 7-10, only 2 are preceded by an "A"!

These are the first two A's in BWT

⇒ 1st two A's in sorted/suffix order


2 3 4 5
BWT G G G G G G T C A A $ T A A
Sorted text $ A A A A C G G G G G G T T
A
1 2
G G G G G G T C A A $ T A A
$ A A A A C G G G G G G T T
G
2 3
G G G G G G T C A A $ T A A
$ A A A A C G G G G G G T T
A
G G G G G G T C A A $ T A A
$ A A A A C G G G G G G T T
G
Reversed query
13 6 8 10 1 4 12 5 7 9 0 3 11 2

So, must be at position 1&2!

Then, "AGA" preceded by "G".

Both 1 & 2 are, use sorted order

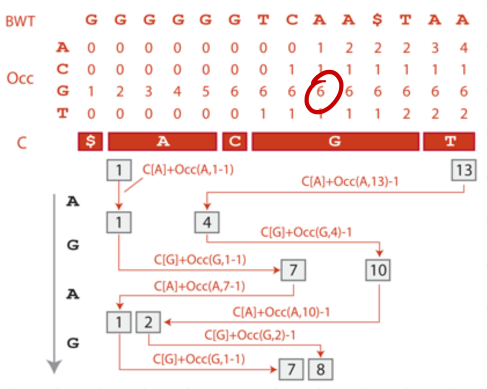
⇒ position 7 & 8 match

Implementation:

Need first & Last row

Sorted → first; BWT → Last

Plus the index counting # of occurences:



BWT		G	G	G	G	G	G	T	C	A	A	$	T	A	A
Occ	A	0	0	0	0	0	0	0	0	1	2	2	2	3	4
	C	0	0	0	0	0	0	0	1	1	1	1	1	1	1
	G	1	2	3	4	5	6	6	6	6	6	6	6	6	6
	T	0	0	0	0	0	0	1	1	1	1	1	2	2	2

Space: For OCC: one row per alphabet character = σ or |Σ|

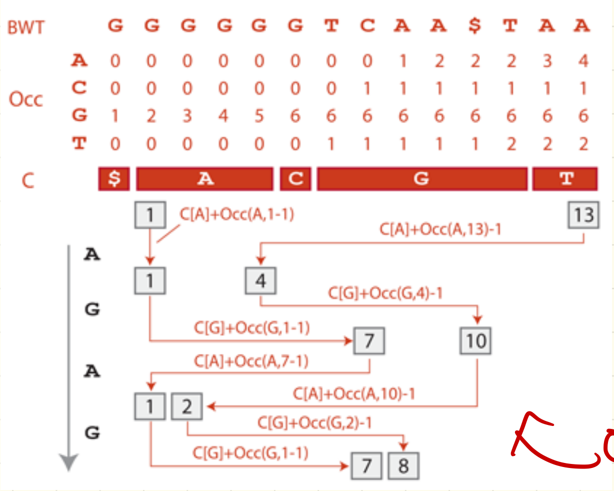
+ one column per input string character = N

Each entry stores log N bits

Total: $O(\sigma N \log N)$ (uncompressed)

For human genome - this was 47.68 GB

Searching:



BWT		G	G	G	G	G	G	T	C	A	A	$	T	A	A
Occ	A	0	0	0	0	0	0	0	0	1	2	2	2	3	4
	C	0	0	0	0	0	0	0	1	1	1	1	1	1	1
	G	1	2	3	4	5	6	6	6	6	6	6	6	6	6
	T	0	0	0	0	0	0	1	1	1	1	1	2	2	2

O(k)

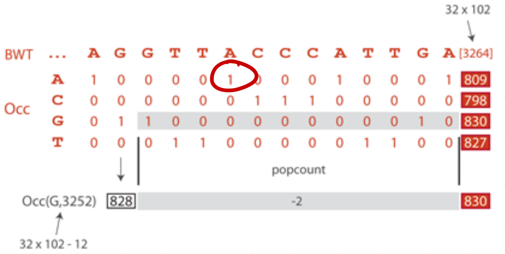
For query of size k:
k steps, each with 2 memory accessed

Note: independent of size of the text!!

O(k) time

Space improvements:

Store 0/1 count (instead of lg N bits)



		...														
BWT		...	A	G	G	T	T	A	C	C	C	A	T	T	G	A [3264]
	A		1	0	0	0	0	1	0	0	0	1	0	0	0	1	809
Occ	C		0	0	0	0	0	0	1	1	1	0	0	0	0	0	798
	G		0	1	1	0	0	0	0	0	0	0	0	0	1	0	830
	T		0	0	0	1	1	0	0	0	0	0	1	1	0	0	827

Keep 1 column per 32, & then just count using binary table.

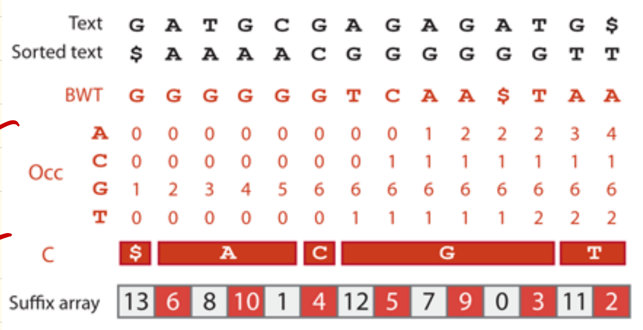
Now: σN bits (plus lg N for every 32nd entry)

(For human genome, now down to 2.98 GB, not 47.68 GB)

Also - compress the suffix array:

Keep 1 value out of every 32

How to compute missing values?



Text		G	A	T	G	C	G	A	G	A	G	A	T	G	\$
Sorted text		\$	A	A	A	A	C	G	G	G	G	G	G	T	T
BWT		G	G	G	G	G	G	T	C	A	A	\$	T	A	A
Occ	A	0	0	0	0	0	0	0	0	1	2	2	2	3	4
	C	0	0	0	0	0	0	0	1	1	1	1	1	1	1
	G	1	2	3	4	5	6	6	6	6	6	6	6	6	6
	T	0	0	0	0	0	0	1	1	1	1	1	2	2	2
C		\$	A				C	G						T	
Suffix array		13	6	8	10	1	4	12	5	7	9	0	3	11	2

Cool trick!

- \$ is stored at 0 & contains value 13 & letter G

Where is 12?

$C[G] + occ(G,0) - 1$
$= 6 + 1 - 1 \Rightarrow$ position of 12!

Generally: if y stored at m, $BWT[m] = x$, $y - 1$ is at $C[x] + occ(x,m) - 1$

If we do this:

Just iterate this: compute position of previous suffix until you reach a multiple of 32

+ look up those values.

(2 memory access per iteration, + at most 31 iterations to reach multiple of 32)

Space:

+: Saves another factor of 32.

For human genome, now down to ~300 MB or so.

(Even more tricks using advanced data structures - bit beyond our scope)

Most famous application:

Seeding step of DNA alignment

BWA uses exact tricks we just looked at.

Particularly good in biology, since "alphabet" is so small.